

1 **BYTE STREAM ORGANIZATION WITH IMPROVED RANDOM AND KEYED**
2 **ACCESS TO INFORMATION STRUCTURES**

3 **FIELD OF THE INVENTION**

4 The invention is directed to improving the performance of systems that access
5 information selectively from information structures organized as byte streams. These include
6 content-based publish/subscribe messaging and database systems.

7 **BACKGROUND OF THE INVENTION**

8 Content-based publish/subscribe messaging requires access to arbitrary message fields in
9 each network node in order to route messages. Messages arrive as byte streams and only a few of
10 the message's fields need to be accessed. However, the fields used to make routing decisions
11 may be anywhere in a complex structured message. The property of *random access* to fields in a
12 byte stream enables routing decisions to be organized optimally without regard to the order in
13 which information is extracted from the byte stream, and it completely avoids any overhead
14 associated with parsing information that isn't needed. The same property of random access to
15 information structures stored in a byte stream form is useful in other systems as well, for
16 example, database systems.

17 It is well known that untagged binary formats can provide constant time random access to
18 fields in a byte stream by using offset calculations (perhaps indirected through offsets stored in
19 the byte stream). However, this only works when the information structure is "flat" (does not
20 involve any nesting of information). In practice, most information structures are not flat.

21 An information structure with a flat structure may be characterized as a *tuple* (or
22 "structure" or "record"). The schema for such an information structure calls for a fixed sequence

1 of fields. In this description, we use the notation [... , ... , ...] for tuple schemas. So, [**int** ,
2 **string** , **boolean**] might be the schema for an information structure containing an integer
3 followed by a string followed by a boolean.

4 Ways in which information structures such as messages nest information and therefore
5 deviate from flatness include at least the following.

6 Tuples may be nested. That is, the schema for an information structure might be [**int** , [**int** , [**string** , **boolean**]]].

8 Any schema element may be repeated zero or more times, forming a *list*. In this
9 description, we use the notation **(...)** for a list in a schema. So, **(int)** means a list of zero or
10 more integers. A list of tuples (often called a "table" or "relation") is also possible. So, **([int*,
11 *string*, *boolean])** is the schema for a table with three columns (an integer column, a string
12 column and a boolean column) and zero or more rows. In most relational databases, each row is
13 a flat structure. But, in messages and advanced databases, each row may have nested tuples and
14 embedded tables, with no intrinsic limit to how deep such nesting can go. Tuples and lists must
15 be allowed to nest in arbitrary ways to accurately describe information structures in general.

16 Information structures may be recursive. For example, a field of a tuple may be defined
17 as another instance of the tuple itself or of an encompassing tuple or list (this cannot be
18 illustrated readily with the present notation).

19 Information structures may include variants in additions to tuples and lists. A variant
20 indicates that either one type of information *or* another (not both) may appear. Information
21 structures may also include dynamically typed areas in which *any* kind of information may
22 appear.

23 It is common to define certain columns of a table as *key* columns. A lookup in the
24 information structure requires finding a particular value in a particular column of the table, after
25 which only that row (or only a specific field from the row) is accessed. In a database, an index
26 might be built in order to do this efficiently. In messages, the tables are rarely large enough to
27 benefit from a precomputed index, and transmitting such an index in the message adds
28 unacceptable overhead. So, for utility in the messaging domain a processor should be able to

1 *scan* just the key column (sequentially) and then randomly access just the information in its row.

2 In addition to what is known about using offset calculations to provide constant time
3 access to completely flat information structures like [**int**, **string**, **boolean**] said techniques are
4 readily extended to encompass just nested tuples (with no lists) such as [**int**, [**int**, **string**, [**string**, **boolean**]]] (by treating it as if it were [**int**, **int**, **string**, **string**, **boolean**]. This is
5 what is done, for example, in an optimizing compiler when compiling code for nested struct
6 declarations in (for example) the C language.

7 A tuple containing fields of varying length requires some pointer indirection in order that
8 all the offsets are still known. For example, if **int** and **boolean** have a fixed-length
9 representation but string does not, then we might represent the two string values in [**int**, **int**,
10 **string**, **string**, **boolean**] as fixed-length pointers to strings stored elsewhere in memory. That
11 way, the last two fields of the tuple are still at a fixed distance from its start (which is how
12 programming languages solve the problem). It is well-known that a pointer to elsewhere in
13 memory can be represented as a stored offset to elsewhere in a byte stream. So, this issue is
14 solvable for byte streams as well as computer memories. Solutions like this are embodied in
15 many Internet protocols to speed up access to information following a varying length field.

16 A simple table (where each row is flat because there are no nested tuples or lists) can be
17 stored in either row order or column order. Varying the storage order for simple
18 multi-dimensional arrays is a well-known technique for optimizing compilers. Relational
19 databases often store tables in column order, since this can improve scan time for key columns
20 that lack indices. However, in messaging, the representation is usually a tree structure and
21 serialization of messages is done by recursive descent, which results in storing all tables in row
22 order. In any case, the well-known technique of storing tables in column order must be extended
23 in non-obvious ways to be useful when schemas use arbitrary nesting of lists within tuples within
24 lists.

25 Schemas whose structure is inconvenient can sometimes be transformed into isomorphic
26 schemas that are more convenient. The flattening of [**int**, [**int**, **string**, [**string**, **boolean**]]]
27

1 to [**int** , **int** , **string** , **string** , **boolean**] is an example of one such isomorphism. The same kind
2 of flattening can be applied to variants. It is also known to those skilled in the field of type
3 theory that tuples can be distributed over variants to yield an isomorphic schema. If we use the
4 notation { **int** | **boolean** } to mean the variant whose cases are **int** or **boolean**, then [**string**, { **int** |
5 **boolean** }] is isomorphic to { [**string** , **int**] | [**string** , **boolean**] }. This observation has been
6 used to improve message processing time in IBM web sites employing the Gryphon system since
7 2001, and also in the IBM Event Broker product.

8 SUMMARY OF THE INVENTION

9 The invention improves access time for elements of lists in randomly accessing a byte
10 stream, particularly when lists represent tables with key columns. It works with information
11 structures whose schemas contain arbitrarily nested tuples and lists. It works in the presence of
12 other information structure elements such as variants, recursion, and dynamic typing, although its
13 improvements are focused on lists.

14 The invention stores tables in *nested column order*, extending the concept of column
15 order so as to apply to arbitrarily nested tables. By using standard offset calculation techniques
16 within the nested lists that result from nested column order, the invention makes both sequential
17 scanning and random access (by row position) efficient. Thus, the problem of finding row
18 contents corresponding to a specific value of a key column is rendered efficient and this extends
19 to nested cases.

20 We first describe the invention with reference to the following example schema, using the
21 notation introduced earlier.

22 ***([string , *([string , int])*])***

23 This schema describes information structures each is comprised of a table with two
24 columns. The first column contains string values, but the second column contains "table" values.
25 Each table appearing as a value in the second column is itself a table of two columns, a string

column and an int column. For example, one might have an information structure conforming to this schema whose logical structure is as follows.

"ages"	"john"	22
	"mary"	14
	"bill"	32
""temperatures"	"arizona"	89
	"alaska"	27

Looking only at the schema, we see that it contains three entries of scalar type, which could be labeled as follows:

```
* ( [ string , *( [ string , int ] ) * ] ) *
```

1 2 3

Each of these entries will give rise to exactly one encoding area in the byte stream. The start of each encoding area will be known by storing its offset in the byte stream at a known offset from the beginning. Area 1 is a list of strings. In the serialization of the example table it would contain

"ages" "temperatures"

(an example of column order as usually understood).

Area 2 is a list of lists of strings. In the serialization of the example table it would contain

("john" "mary" "bill")("arizona" "alaska")

This is an example of nested column order.

Area 3 is a list of lists of int. In the serialization of the example table it would contain

(22 14 32)(89 27)

(another example of nested column order).

The byte encoding for a list comprises the number of elements in the list followed by an encoding that depends on whether the fields have fixed or varying length. Fixed length fields are just stacked immediately after each other, since the offset to any one of them can be computed by

1 multiplying the index position by the length of each field. For example, if an int requires four
2 bytes to encode, then the list (22 14 32) can be efficiently encoded in 12 bytes. The first element
3 is at offset 0, the second at offset 4, the third at offset 8 and there is no need to record any offsets
4 in the byte stream.

5 However, varying length fields require offsets to be recorded. Thus, the list ("john"
6 "mary" "bill") would be recorded as a table of offsets to the actual elements. Since the offset
7 entries have fixed length, these can be randomly accessed. The offset table is followed by the
8 elements themselves (strings in this case). These elements can be scanned sequentially as well as
9 indexed randomly when accessing information from the byte stream.

10 A list of lists is just a special case of this varying length list byte encoding. Each list is
11 treated like a varying length value in forming the overall list.

12 Consider how the invention helps with a efficient keyed access. Suppose the problem is
13 to access the table of "temperatures" from the byte stream and then look up the temperature in
14 "arizona." Area one can be scanned sequentially with high efficiency, determining that the
15 "temperatures" row in the table is the second row. We then access the second element in the
16 second area randomly, finding the offset of its value, which is ("arizona" "alaska"). Scanning
17 this sequentially, we find that "arizona" is the first element. We then access the third area with
18 the successive indices just computed and go quickly to the highlighted element in (22 14 32)(89
19 27). Only the desired value (89) is actually deserialized from the byte stream.

20 The invention accommodates dynamically typed information by treating dynamically
21 typed areas as if they were scalars. The invention can be employed recursively to encode the
22 dynamically typed areas. The invention accommodates recursive schemas by treating
23 self-referential areas of the schema as if they were dynamically typed areas with a completely
24 new schema and then using itself recursively to encode said areas. The invention accommodates
25 variants by treating them as if they were dynamically typed areas, if necessary. The invention can
26 be employed recursively to encode the variant case that is actually present in the message after
27 recording a tag that indicates which case is present. The type isomorphism that allows tuples to
28 be distributed over variants can also be employed to move as many variants as possible to the

1 top-level of the schema, which maximizes the invention's scope for producing highly efficient
2 results.

3 **BRIEF DESCRIPTION OF THE DRAWINGS**

4 The invention and its embodiments will be more fully appreciated by reference to the following
5 detailed description of advantageous but nonetheless illustrative embodiments in accordance with
6 the present invention when taken in conjunction with the accompanying drawings., in which:

7 Fig. 1 shows an example of a tree representation of a schema as required by the invention..

8 Fig. 2 shows the schema of Fig. 1 with the nodes augmented to show some aspects of a byte
9 stream layout. The computation of a layout is the first of three processes that constitute the
10 invention.

11 Fig. 3 shows one way of completing a layout for the example schema.

12 Fig. 4 shows an alternative way of completing a layout for the example schema.

13 Fig. 5 shows the sub-process structure of the serialization process, which is the second of the
14 three processes constituting the invention.

15 Fig. 6 shows a the in-memory representation of an information structure as a tree, where the
16 information structure conforms to the example schema. It is used to guide a detailed example of
17 the serialization process.

18 Fig. 7 shows the result of applying the serialization process to the information structure of
19 Fig. 6.

20 Fig. 8 shows a different example schema that benefits from reorganization in order to achieve
21 maximum benefit from the invention.

22 Fig. 9 shows the example schema of Fig. 8, reorganized so as to achieve maximum benefit
23 from the invention.

DETAILED DESCRIPTION OF THE INVENTION

The invention assumes that information structures are described by *schemas*, which is a common practice. Schemas can be represented in computer memory as rooted directed graphs whose leaf nodes represent scalar data types or dynamic type (the latter meaning that any type of information may be present) and whose interior nodes represent data structures such as **tuples**, **lists**, and **variants**. A **list** node has exactly one child, a **tuple** or **variant** node has one or more children. Cycles in the graph represent recursive definitions in the schema; a non-recursive schema's graph representation will be a tree.

To employ the invention, the schema's graph representation must be simplified to a tree representation by truncating recursive definitions and replacing them with dynamic type (which is represented by a leaf node in the truncated schema). The invention may then be employed recursively to serialize the recursive definition as if it were of dynamic type.

To employ the invention, the schema's tree representation must be made free of variants. Variants may be changed to dynamic type, which become leaves of a truncated schema as with recursive definitions. The invention may then be employed recursively to serialize the particular case of the variant that arises in the information structure as if it were of dynamic type. An optional schema reorganization process is presented later in this description showing how the effectiveness of the invention can be improved in the presence of variants.

An example of a schema tree representation is shown in Figure 1. The arrows of Figure 1 show the classic parent-child relationship that is present in any tree. Nodes 1 through 6 are *interior* nodes representing **tuples** (nodes 1, 3, 4, and 6) or **lists** (nodes 2, and 5). Nodes 7 through 14 are *leaf nodes* representing integers (nodes 8, 10, and 13), strings (nodes 7, 9, 11, and 12), and booleans (node 14 is the sole case in this example). The example does not include any dynamic type leaf nodes.

1 To use the invention, a repertoire of scalar data types is chosen that will be accepted as leaf
2 schema nodes. Once that choice is made, the invention supports *all* possible schemas made up of
3 **list** nodes, **tuple** nodes, and the scalar data types plus dynamic type as leaf nodes. As noted
4 above, such schemas can be derived from more complex schemas containing recursion and
5 variants by truncating the schema and replacing the truncated portions with dynamic type leaves.

6 To use the invention, a method is chosen to encode values of each scalar data type as a
7 sequences of bytes. Each such byte encoding should have fixed length if that is both possible and
8 efficient, a variable-length encoding otherwise.

9 If a variable length byte encoding is employed, the encoding should provide a way of
10 knowing where one encoded item ends and the next begins. A standard technique (not
11 necessarily the only one) is to begin a variable length encoding with its length.

12 To use the invention, a method is chosen to encode schema tags that efficiently denote
13 schemas. Information of dynamic type is encoded by encoding its schema tag and then using the
14 invention recursively to encode the information of dynamic type. The resulting encoding is
15 necessarily a variable length encoding.

16 In this description, only **int**, **string**, and **boolean** scalar data types are mentioned but the
17 invention is not restricted to that scalar data type repertoire, or any other. This embodiment
18 employs fixed length byte encodings for **int** and **boolean** values that are four bytes and one byte,
19 respectively, and a variable length encoding for **string** values. These choices are not essential to
20 the invention.

21 The invention also requires a way of encoding non-negative integers that are used as lengths
22 and offsets. In this embodiment we employ four-byte big-endian integers. However, that
23 particular choice is not essential to the invention.

24 The invention assumes that all information structures to which the invention will apply have
25 an in-memory representation which is a tree. For each node in the in-memory representation, it is
26 possible to find a corresponding node in the schema tree representation that gives its type.

1 The invention comprises three interrelated processes, which together, deliver the goal of
2 efficient random and keyed access to byte stream contents. A fourth, optional, process may be
3 used to reorganize a schema containing variants so as to more effectively exploit the invention.

4 1. A process for computing a *layout* from a schema tree representation. A layout guides the
5 serialization of all information structures conforming to said schema. Serialization (a term
6 familiar to those skilled in this art) means the formation of the byte stream from the in-memory
7 representation. The layout computation need be done only once for each schema and need not
8 (should not) be redone each time an information structure conforming to that schema is
9 serialized.

10 2. A process for serializing the byte stream, input to comprise of the layout and the
11 in-memory representation, output to comprise of a byte stream. The serialization process occurs
12 only when an in-memory representation exists and a byte stream representation of the same
13 information is desired. In messaging, for example, this would happen in computers that originate
14 messages and would not typically be redone in computers that are merely routing the message.

15 3. A process for efficiently accessing any scalar value from within the serialized byte stream
16 without deserializing surrounding parts of the byte stream (to those skilled in this art,
17 deserialization means forming an in-memory representation from a byte stream, the inverse of
18 serialization). The access process provides the real benefit of the invention, but the other two
19 processes are necessary in support of this goal.

20 4. (Optional) A process for reorganizing a schema into possibly several schemas so that the
21 number of variants to be changed to dynamic type is reduced and the effectiveness of the
22 invention is increased.

23 The rest of this description covers the three processes plus the fourth optional process.

24 THE LAYOUT COMPUTATION PROCESS

25 The process for computing the layout comprises three steps.

26 **Step 1.** Assign two values defined as follows to each leaf node in the schema. Both values
27 can be assigned in a single depth-first left-right traversal of the schema.

1 1. A consecutive increasing *field number* is assigned to each leaf node encountered, in
2 depth-first left-right order.

3 2. A *path* is assigned to each leaf node showing the sequence of interior nodes that reaches
4 that leaf from the root of the schema. If every node has a distinct machine address (as is usual
5 with this form of representation), it is sufficient to record the sequence of machine addresses that
6 constitute the path.

7 Figure 2 shows the schema of Figure 1 after step one of the layout computation process has
8 been carried out. The interior nodes labeled 18 through 23 are assigned unique letter codes A
9 through F only to show that each has a unique machine address. The field numbers assigned by
10 this step in the process are shown as unbracketed numbers in each leaf node in Figure 2. The
11 field numbers 0 through 7 have been assigned to the nodes labeled 24 through 31, respectively.
12 The paths are shown as bracketed sequences of letters representing machine addresses of interior
13 nodes. For example, the node labeled 28 has the path [ABCDE] because the path to that node
14 starting with the root encompasses exactly the interior nodes 18 through 22 and the letters A
15 through E represent their machine addresses, respectively.

16 **Step 2.** Construct a template (hereafter called the *layout*) that will apply to the byte stream
17 form of every information structure conforming to the schema. The layout calls for two byte
18 stream *portions*, a *fixed length portion*, to come first, and a *variable length portion*, to follow
19 immediately thereafter.

20 The fixed length portion always has a predictable length, and is divided into *slots*, each of
21 which has a predictable length. Thus, every slot in the fixed length portion is at a known offset
22 from the start of the byte stream. The start of the variable length portion is at a known offset
23 from the start of the byte stream, but offsets to information within the variable length portion
24 may vary among byte streams using the same layout, depending on the number of bytes occupied
25 by byte encodings earlier in the variable length portion. .

1 This description provides two alternative *styles* that layouts may follow. The first represents
2 a well-known approach to handling mixtures of fixed-length and varying length fields similar to
3 what is done by compilers for programming languages and by some DBMSs. The second style is
4 equally efficient and has the occasionally useful property that the fixed length portion contains
5 only offsets instead of a mixture of offsets and data.

6 **Layout Style 1.** In this style, the fixed length portion of the byte stream has one fixed length
7 slot for each field number computed in step 1, in order of ascending field numbers.

8 For each leaf node whose paths contain no **lists** and whose scalar data type has a fixed length
9 encoding, information corresponding to that leaf node will be serialized directly into the fixed
10 length slot corresponding to that node's field number.

11 For each leaf node whose paths contain **lists** *and/or* whose scalar data type has a variable
12 length encoding, the fixed length slot corresponding to that node's field number will contain an
13 offset into the variable length portion and information corresponding to that leaf node will be
14 serialized at that point in the variable length portion.

15 Figure 3 shows the layout constructed in style 1, using the example schema of Figure 1 as
16 numbered in Figure 2. Slots 32 through 39 correspond to field numbers 0 through 7,
17 respectively. Fields 1 and 6 are fixed length and hence will be encoded in their slots (33 and 38)
18 in the fixed length portion. The other fields will be encoded in the variable length portion, with
19 field 0 serialized as shown in a range of bytes labeled 40, field 2 in a byte range labeled 41, field
20 3 in a byte range labeled 42, field 4 in a byte range labeled 43, field 5 in a byte range labeled 44,
21 and field 7 in a byte range labeled 45. The arrows labeled 46 through 51 represent offsets. The
22 slots labeled 32, 34, 35, 36, 37, and 39 contain the offsets of the beginnings of byte ranges
23 labeled 40, 41, 42, 43 44, and 45, respectively.

1 **Layout Style 2.** All of the fields numbered in step 1 will be encoded in the varying length
2 portion, consecutively in ascending order of field number, no matter whether the fields are of
3 fixed or varying length. The fixed length portion will contain one offset slot for each field that
4 *follows* a varying length field. The fixed length portion comprises *only* of these needed offset
5 slots. For this purpose, a field has varying length if its path contains *lists and/or* the scalar data
6 type from the schema has a variable length encoding.

7 Figure 4 shows a layout constructed in style 2, using the same example as with previous
8 Figures. Fields 1, 3, 4, 5, and 6 follow fields with variable length encodings, hence they require
9 offset slots in the fixed length portion of the byte stream. The five offset slots labeled 52
10 through 56 are reserved in the fixed length portion. The variable length portion contains byte
11 ranges labeled 57 through 64, containing the serialized information for fields 0-7 respectively.
12 Field 0 doesn't follow any field, and fields 2 and 7 follow fixed length fields. Hence, their offsets
13 are not be recorded in the fixed length portion. They either have a known offset already or their
14 offset can be calculated from the offset of a previous field (which is recorded). The arrows
15 labeled 65 through 69 convey the fact that the offsets in slots labeled 52 through 56 are to the
16 beginnings of the byte ranges labeled 58, 60, 61, 62, and 63, respectively, in which are serialized
17 the information for fields 1, 3, 4, 5, and 6, respectively.

18 **Step 3.** The information computed in the two previous steps is organized for efficient
19 lookup by field number. That is, given a field number, one can quickly find its leaf node in the
20 schema (hence its data type and path) and also its place in the layout. This can be done by
21 recording appropriate machine addresses and other information in an array indexed by field
22 number.

23 **THE SERIALIZATION PROCESS**

24 The process for forming a byte stream from an in-memory representation is called
25 serialization. The serialization process has a sub-process structure shown in Figure 5.

1 The serialization master sub-process (labeled 70) sequences the process as a whole. It
2 invokes, as needed, (1) a sub-process for non-list values (labeled 72) and (2) a sub-process for
3 list values (labeled 74).

4 The list sub-process invokes, as needed, (1) a sub-process for fixed length items (labeled 76) ,
5 (2) a sub-process for variable length items that are not lists (labeled 78), and (3) a sub-process for
6 nested list items (labeled 80), which, in turn, recursively invokes the list sub-process.

7 The directed arrows 71, 73, 75, and 77 indicate invocation of one sub-process by another,
8 with the sub-process at the point of the arrow returning eventually to its invoker. The
9 bi-directional arrow 79 indicates that the processes labeled 74 and 80 can invoke each other.
10 However, each such invocation eventually returns to the invoking process.

11 Our description of the serialization process describes in-memory representations of
12 information structures, and then details the steps carried out by each sub-process

13 **In-memory representations of information structures.**

14 In any application of this invention, there will be some representation for information
15 structures in computer memory.

16 Since information structures conform to tree-like schemas, we assume that a tree-like
17 in-memory representation of those information structures is always possible. In the case of
18 messages, tree like representations are the norm.

19 For example, the DOM standard from W3C, or the JAXB standard from Javasoft, specify
20 tree-like representations, as does the SDO representation proposed by IBM and BEA. The details
21 of the representation are unimportant, but all such representations have common elements.

22 All in-memory representations conforming to a particular schema will have nodes whose
23 types correspond to nodes in said schema's tree representation, as follows.

24 1. Scalar values in the in-memory representation and dynamically typed nodes correspond to
25 leaf schema nodes designating the data type of the value. For example, an integer 3 might
26 correspond to an **int** schema node, the string "charles" might correspond to a **string** schema
27 node, the truth value **false** might correspond to a **boolean** schema node, etc.

1 2. Heterogeneous container nodes (such as Java beans) in the in-memory representation
2 correspond to **tuple** nodes in the schema. For example, a bean with **int** and **string** fields
3 corresponds to a **tuple** in the schema whose children are **int** and **string** leaf nodes.

4 3. Homogeneous container nodes (such as lists, arrays, or sets) in the in-memory
5 representation correspond to **list** nodes in the schema. For example, an array of strings
6 corresponds to a **list** node in the schema whose child is a **string** leaf node.

7 Correspondence is, in general, many-to-one. That is, more than one node in the in-memory
8 representation can correspond to the same node in the schema.

9 Some tree representations (e.g. DOM) don't clearly distinguish between homogeneous and
10 heterogeneous collections. However, when the schema is available, the distinction can be
11 reconstructed (see **step 1** of the **serialization master sub-process**).

12 Figure 6 shows an in-memory representation of an information structure that conforms to the
13 schema introduced in Figures 1 and used in the layout computation examples of Figure 2 through

14 4. The nodes in Figure 6 are annotated with the parenthesized field numbers (for scalar data
15 values) or node letters (for **bean** and **array** nodes) taken from Figure 2. This shows the
16 correspondence between nodes of an in-memory representation and the nodes in its schema tree
17 representation. More precisely characterizing the nodes in Figure 6:

18 the node labeled **81** corresponds to the node labeled **18** in Figure 2, as indicated by the address
19 (A);

20 the node labeled **82** corresponds to the node labeled **19** in Figure 2, as indicated by the address
21 (B);

22 the nodes labeled **83** and **84** correspond to the node labeled **20** in Figure 2, as indicated by the
23 address (C) recorded in both;

24 the nodes labeled **85** and **86** correspond to the node labeled **21** in Figure 2, as indicated by the
25 address (D) recorded in both;

26 the nodes labeled **87** through **90** correspond to the node labeled **22** in Figure 2, as indicated by
27 the address (E) recorded in all of them;

1 the node labeled **91** corresponds to the node labeled **23** in Figure 2, as indicated by the address
2 (**F**);
3 the node labeled **92** corresponds to the node labeled **24** in Figure 2, as indicated by the field
4 number (**0**);
5 the node labeled **93** corresponds to the node labeled **25** in Figure 2, as indicated by the field
6 number (**1**);
7 the nodes labeled **94** and **95** correspond to the node labeled **26** in Figure 2, as indicated by the
8 field number (**2**) recorded in both;
9 the nodes labeled **96** through **99** correspond to the node labeled **27** in Figure 2, as indicated by
10 the field number (**3**) recorded in all of them;
11 the nodes labeled **100** through **103** correspond to the node labeled **28** in Figure 2, as indicated by
12 the field number (**4**) recorded in all of them;
13 the nodes labeled **104** through **107** correspond to the node labeled **29** in Figure 2, as indicated by
14 the field number (**5**) recorded in all of them;
15 the node labeled **108** corresponds to the node labeled **30** in Figure 2, as indicated by the field
16 number (**6**); and
17 the node labeled **109** corresponds to the node labeled **31** in Figure 2, as indicated by the field
18 number (**6**).

19 The term **bean** in the diagram should be understood to represent any heterogeneous
20 container, not necessarily literally a Java bean. The term **array**, similarly, represents any
21 homogeneous collection that supports a determination of its size and iteration over its elements.
22 Different in-memory representations will use different object types to represent aspects of the
23 information structure but the invention applies to all possible choices of representation using
24 parent-child relationships, and heterogeneous or homogeneous collections, as described herein.

25 **The Serialization master sub-process.**

26 The serialization master sub-process labeled **70** in Figure 5 comprises 6 steps.

1 **Step 1.** The correspondence between the in-memory representation and its schema tree
2 representation is determined.

3 In some cases (for example, SDO), this correspondence is given *a priori* (nodes in the
4 in-memory representation are specialized objects that point to their schema node).

5 In other cases (for example DOM), this correspondence can be computed by available tools
6 (the XML Schema standard from W3C defines *validating parsers* which, in addition to
7 validating that an information structure conforms to its schema, compute a *post-validation info* set
8 that makes explicit the correspondence of elements of the DOM tree to the schema).

9 Figure 6 can be taken as illustrating the end point of this step, in which the parenthesized
10 letters denoting the machine addresses of interior schema nodes and the parenthesized field
11 numbers denoting the addresses of leaf schema nodes show the correspondence to the schema in
12 Figure 2.

13 For the remaining steps, we assume that the schema is available. If the layout computation
14 has not been performed on the schema before this moment, it is performed now. For the
15 remaining steps, we assume that a layout is available governing all byte streams that serialize
16 information structures conforming to this schema.

17 **Step 2** Using the layout, the byte stream is initialized by reserving the fixed length portion
18 (assigning memory to it without yet specifying its contents) and recording the address of the
19 beginning of the variable length portion, which will grow by appending bytes to the end. This
20 pointer to the beginning of the variable length portion is called the *current encoding point* and
21 will be incremented by other steps so as to always point to the end of material that has already
22 been encoded in the variable length portion.

23 **Step 3.** The sub-process iterates through the field numbers (in increasing order) that were
24 assigned in the layout computation process. Steps 4 and 5 are carried out for each field number.

25 **Step 4 (repeated by step 3).** The location for the field in the byte stream is looked up in the
26 layout. If that location is in the variable length portion, the place to encode the field is always the
27 *current encoding point*. This assumption is sound because both layout styles encode varying
28 length fields in increasing field number order.

1 If that location is in the variable length portion, and the layout calls for its offset to be
2 recorded in a fixed length slot, then the current encoding point is converted to an offset from the
3 start of the variable length portion and that offset is stored in the fixed length slot as called for by
4 the layout.

5 **Step 5 (repeated by step 3).** If the field's path contains no **list** node, the single scalar value
6 in the in-memory representation for that field is encoded by the **non-list sub-process** (arrow 71
7 to box 72 in Figure 5). Otherwise, the potentially many scalar values for the field are encoded by
8 the **list sub-process** (arrow 73 to box 74 in Figure 5) Both of these subprocesses increment the
9 current encoding point.

10 **Step 6.** The final value of the current encoding point, minus the start of the byte stream,
11 gives the length of the byte stream. Serialization is complete.
12 **(end of master sub-process).**

13 **The Non-list sub-process.**

14 This sub-process is invoked for a particular field in the layout that has no **list** in its path. The
15 layout will dictate where in the byte stream the single scalar value for the field should be
16 encoded, as determined in the master sub-process, step 4. This will either be a fixed length slot
17 or the current encoding point. The sub-process has three steps.

18 **Step 1.** The path computed as part of the layout indicates how to navigate through the
19 in-memory representation to find the scalar value to be encoded. Since there are no **list** nodes in
20 the path, this navigation yields a single value. The scalar value to be encoded is accessed by
21 following the path through the in-memory representation.

22 **Step 2.** The value found in step 1 is encoded into the byte stream location that was
23 determined in by the master sub-process, step 4. The encoding algorithm for all supported scalar
24 data types was supplied by the user of the invention and is not intrinsic to the invention.

25 **Step 3.** If the current encoding point was used, it is incremented by the number of bytes it
26 took to perform the encoding in **step 2**.

27 **(end of non-list sub-process)**

1 **The List sub-process.**

2 The list sub-process can be invoked by the master sub-process (arrow 73 in Figure 5) or
3 recursively via the **list-valued item sub-process** as indicated by the two-way arrow labeled 79 in
4 Figure 5. In addition to being invoked for a particular field in the layout, each invocation has, as
5 an argument, a *starting node* within the in-memory representation. When the list sub-process is
6 invoked from the master sub-process, the starting node is the root node of the in-memory
7 representation. When the sub-process is invoked from the list-valued item sub-process, the
8 starting node will be an interior node within the in-memory representation whose parent is a
9 homogeneous collection.

10 No matter the layout style, all lists will be encoded in the variable length portion of the byte
11 stream because lists have intrinsically varying length.

12 The list sub-process has 6 steps.

13 **Step 1.** Increment the current encoding point by four bytes to leave room to record the
14 overall length of the list (to be recorded in step 6). The previous value of current encoding point
15 is remembered as **lengthLocation** and the new value as **sizeLocation**. Note: recording the
16 length of every list at the start of the list is done so that the start and end of each list will be
17 unambiguous when scanning the byte stream, which is considered good practice. No part of the
18 invention as described herein actually requires this to be done, so this step is not intrinsic to the
19 invention.

20 **Step 2.** Find the schema node within the field's path that corresponds to the starting node.
21 This will be the first node when the list sub-process is invoked from the master sub-process. It
22 will be a node immediately following a **list** node in the path otherwise.

23 **Step 3.** Navigate the in-memory representation from the starting node, using a suffix of the
24 path starting at the node found in **step 2**, until the next homogeneous collection node in the
25 in-memory representation (corresponding to a **list** node in the path) is encountered. Record the
26 *residual path* which is the part of the path after the **list** node that was matched in this step. The
27 residual path may be empty.

1 **Step 4.** Determine the size of the homogeneous collection node that was navigated to in **step**
2 **3** (the number of items in the collection). Record this size as a big-endian four-byte integer at
3 the current encoding point and increment the current encoding point by four bytes.

4 **Step 5.** Perform one of three possible actions.

5 1. If the residual path from **step 3** contains any **list** node, then perform the **list-valued item**
6 **sub-process** (arrow 79 to box 80 in Figure 5).

7 2. Otherwise (the residual path contains no **list** nodes), if the scalar data type of the field
8 requires a variable length encoding, perform the **variable-length item sub-process** (arrow 77 to
9 box 78 in Figure 5).

10 3. Otherwise (the residual path contains no **list** nodes *and* the scalar data type of the field has
11 a fixed length encoding), perform the **fixed-length item sub-process** (arrow 75 to box 76 in
12 Figure 5).

13 **Step 6.** Subtract **sizeLocation** from the current encoding point and record the result at
14 **lengthLocation**. This causes the list to be preceded in the byte form by its length in bytes. As
15 noted above, doing is this is not intrinsic to the invention but is considered good practice.
16 **(end of list sub-process)**

17 **The Fixed-length item sub-process.**

18 The fixed-length item sub-process is entered with a homogeneous collection and a residual
19 path, both determined in **step 3** of the **list sub-process**. The fixed-length item sub-process
20 iterates through the collection, performing the following sequence of 3 steps on each item in the
21 collection.

22 **Step 1 (iterated).** The residual path is used to navigate from the item from the collection (a
23 node within the in-memory representation) to a scalar value.

24 **Step 2 (iterated).** The scalar value is encoded at the current encoding point according to its
25 type (in this sub-process that will always be a fixed-length encoding).

26 **Step 3 (iterated).** The current encoding point is incremented by the length of the encoding.
27 **(end of fixed-length item sub-process)**

1 **The Varying-length item sub-process.**

2 This sub-process is entered with a homogeneous collection and a residual path, both
3 determined in **step 3** of the **list sub-process**. The varying-length item sub-process comprises the
4 following 6 steps.

5 **Step 1.** The number of items in the collection is multiplied by four and the current encoding
6 point is incremented by the resulting amount, leaving room for an offset table with as many
7 entries as there are items in the list. Two values called **firstOffset** and **nextOffset** point to the
8 start of this offset table.

9 **Step 2.** The sub-process iterates through the collection, performing the remaining steps on
10 each item.

11 **Step 3 (repeated by step 2).** The current encoding point, minus **firstOffset** is recorded at
12 **nextOffset** and **nextOffset** is incremented by four. This creates an entry in the offset table that
13 was created in step 1.

14 **Step 4 (repeated by step 2).** The residual path is used to navigate from the item of the
15 collection selected by step 2 (a node in the in-memory representation) to a scalar value.

16 **Step 5 (repeated by step 2).** The scalar value is encoded at the current encoding point
17 according to its type (in this sub-process that will always be a variable-length encoding).

18 **Step 6 (repeated by step 2).** The current encoding point is incremented by the length of the
19 encoding produced in step 5.

20 **(end of variable-length item sub-process)**

21 **The List-valued item sub-process.**

22 This sub-process is entered with a homogeneous collection and a residual path, both
23 determined in **step 3** of the **list sub-process**. It comprises four steps, the first three of which are
24 identical to the variable-length item sub-process.

1 **Step 1.** The number of items in the collection is multiplied by four and the current encoding
2 point is incremented by the resulting amount, leaving room for an offset table with as many
3 entries as there are items in the list. Two values called **firstOffset** and **nextOffset** point to the
4 start of this offset table.

5 **Step 2.** The sub-process iterates through the collection, performing the remaining steps on
6 each item.

7 **Step 3 (repeated by step 2).** The current encoding point, minus **firstOffset** is recorded at
8 **nextOffset** and **nextOffset** is incremented by four. This creates an entry in the offset table that
9 was created in step 1.

10 **Step 4 (repeated by step 2).** The **list sub-process** is invoked recursively, with its starting
11 node set to the particular collection item iterated to in step 2. This causes the list to be encoded
12 at the current encoding point and increments the current encoding point appropriately.

13 **Example**

14 The entire byte stream resulting from serializing the in-memory representation in Figure 6 is
15 shown in Figure 7. Layout style 2 (shown in Figure 4) was employed.

16 In Figure 7, the byte stream is shown broken into five rows of information so as to fit the
17 page. In reality, it is a contiguous array of bytes. Each box in the Figure (labeled successively as
18 **200** through **254**) represents some number of contiguous bytes in the byte stream. The text
19 uppermost in each box gives the type of information encoded there and (directly or implicitly) the
20 number of bytes occupied by the information. The text lowermost in each box gives the actual
21 contents, not "byte-by-byte" but in a format designed to facilitate understanding. More
22 specifically,
23 an **O** in a box indicates offset information, said offset information consuming four bytes of the
24 byte stream;
25 an **L** in a box indicates the length of the following list in bytes, said length consuming four bytes
26 of the byte stream (recording list lengths is not essential to the invention but is considered good
27 practice and is done by this embodiment);

1 an **S** in a box indicates a list size (number of items) of a list, said list size consuming four bytes
2 in the byte stream; and

3 a **D** in a box indicates serialized data from the in-memory representation, said data consuming
4 the number of bytes in the byte stream indicated next to the **D**.

5 The following walk-through of the serialization process for this example should make clear how
6 the individual elements of the serialization are generated.

7 **A Walk-through of the Example**

8 This walk-through relates elements of the byte stream shown in Figure 7 to steps in the
9 subprocesses of Figure 5 by walking through the steps, spanning the various subprocesses, by
10 which the in-memory representation shown in Figure 6 was serialized to form the byte stream
11 shown in Figure 7.

12 **Master sub-process begins** (Figure 5, label 70)

13 **Master-1:** relate in-memory representation to its schema (result shown in Figure 6)

14 **Master-2:** initialize layout (reserves the five slots at the start of the byte stream
15 labeled 200 through 204 in Figure 7). These correspond to slots labeled 52 through 56 in
16 Figure 4). Current encoding point (start of variable length portion) is a start of 205 in
17 Figure 7.

18 **Master-3:** iterate through field numbers 0-7 performing steps 4 and 5.

19 **Master-4(0):** field 0 is in variable length portion (box 57 in Figure 4), so serialization
20 is at the current encoding point. No offset to record.

21 **Master-5(0): non-list sub-process** chosen (arrow 71 to box 72 in Figure 5).

22 **Non-list-1:** navigate from node 81 to 92 in Figure 6, based on the path recorded in
23 node 24 in Figure 2. The value is "mary"

24 **Non-list-2:** "mary" appears in result (205 in Figure 7).

25 **Non-list-3:** current encoding point now at start of 206 in Figure 7.

26 **(End Non-list sub-process, return via arrow 71 to box 70 in Figure 5).**

1 **Master-4(1):** field 1 is in variable length portion (box 58 in Figure 4) so serialization
2 is at current encoding point. Offset of box 58 is in box 52 in Figure 4 as shown by arrow
3 65, so the offset of the current encoding point from the start of the variable length portion
4 (8 bytes) is recorded in box 200 in Figure 7.

5 **Master-5(1): non-list sub-process** chosen (arrow 71 to box 72 in Figure 5).

6 **Non-list-1:** navigate from node 81 to 93 in Figure 6, based on the path recorded in
7 node 25 in Figure 2. The value is 63.

8 **Non-list-2:** 63 appears in result (206 in Figure 7).

9 **Non-list-3:** current encoding point now at start of 207 in Figure 7.

10 **(End Non-list sub-process,** return via arrow 71 to box 70 in Figure 5).

11 **Master-4(2):** field 2 is in variable length portion (box 59 in Figure 4), so serialization
12 is at the current encoding point. No offset to record.

13 **Master-5(2): list sub-process** chosen (arrow 73 to box 74 in Figure 5).

14 **List-1:** reserve bytes in box 207 of Figure 7 and set **lengthLocation** to the start of
15 those bytes, current encoding point and **sizeLocation** to the start of box 208.

16 **List-2:** starting node is node 81 in Figure 6.

17 **List-3:** first homogeneous collection in path (see node 26 in Figure 2) corresponds to
18 node 82 in Figure 6.

19 **List-4:** record size of collection (2) in box 208, Figure 7. Current encoding point is
20 now start of 209, Figure 7.

21 **List-5: Variable-length item sub-process** chosen (arrow 77 to box 78 in Figure 5).

22 **Var-item-1:** two-slot offset table reserved (boxes 209 and 210 in Figure 7). Both
23 **firstOffset** and **nextOffset** point to start of 209. Current encoding point is now at start
24 of box 211.

25 **Var-item-2:** iterate remaining steps for the members of the **array** node 82; these are
26 nodes 83 and 84.

1 **Var-item-3(0)**: record offset in slot at **nextOffset**, which is box **209**. The value is
2 current encoding point minus **firstOffset** which is 8 bytes. **NextOffset** is now at the start
3 of box **210**.

4 **Var-item-4(0)**: Residual path is **[C]** (see node **26** in Figure 2), so navigate from node
5 **83** to node **94** in Figure 6. The value is "**charles**"

6 **Var-item-5(0)**: "**charles**" appears in result (**211** in Figure 7).

7 **Var-item-6(0)**: current encoding point is now at **212** in Figure 7.

8 **Var-item-3(1)**: record offset in slot at **nextOffset**, which is box **210**. The value is
9 current encoding point minus **firstOffset** which is 19 bytes. **NextOffset** is now at the
10 start of box **211**.

11 **Var-item-4(1)**: Residual path is **[C]** (see node **26** in Figure 2), so navigate from node
12 **84** to node **95** in Figure 6. The value is "**dog**"

13 **Var-item-5(0)**: "**dog**" appears in result (**212** in Figure 7).

14 **Var-item-6(0)**: current encoding point is now at **213** in Figure 7.

15 **(End Variable-item sub-process, return via arrow 77 to box 74 in Figure 5).**

16 **List-6**: length of list (30 bytes) recorded in at **lengthLocation** (box **207**).

17 **(End List sub-process, return via arrow 73 to box 70 in Figure 5).**

18 **Master-4(3)**: field 3 is in variable length portion (box **60** in Figure 4) so serialization
19 is at current encoding point. Offset of box **60** is in box **53** in Figure 4 as shown by arrow
20 **66**, so the offset of the current encoding point from the start of the variable length portion
21 (46 bytes) is recorded in box **201** in Figure 7.

22 **Master-5(3)**: **list sub-process** chosen (arrow **73** to box **74** in Figure 5).

23 **List-1**: reserve bytes in box **213** of Figure 7 and set **lengthLocation** to the start of
24 those bytes, current encoding point and **sizeLocation** to the start of box **214**.

25 **List-2**: starting node is node **81** in Figure 6.

26 **List-3**: first homogeneous collection in path (see node **27** in Figure 2) corresponds to
27 node **82** in Figure 6.

1 **List-4:** record size of collection (2) in box 214, Figure 7. Current encoding point is
2 now start of 215, Figure 7.

3 **List-5: List-valued item sub-process** chosen (arrow 79 to box 80 in Figure 5).

4 **List-val-item-1:** two-slot offset table reserved (boxes 215 and 216 in Figure 7).

5 Both **firstOffset** and **nextOffset** point to start of 215. Current encoding point is now at
6 start of box 217.

7 **List-val-item-2:** iterate remaining steps for the members of the **array** node 82; these
8 are nodes 83 and 84.

9 **List-val-item-3(0):** record offset in slot at **nextOffset**, which is box 215. The value
10 is current encoding point minus **firstOffset** which is 8 bytes. **NextOffset** is now at the
11 start of box 216.

12 **List-val-item-4(0):** Recursively invoke the **list sub-process** (arrow 79 to box 74 in
13 Figure 7).

14 **(Recursive) List-1:** reserve bytes in box 217 of Figure 7 and set **lengthLocation** to
15 the start of those bytes, current encoding point and **sizeLocation** to the start of box 218.

16 **(Recursive) List-2:** starting node is node 83 in Figure 6.

17 **(Recursive) List-3:** first homogeneous collection in the residual path [CDE] (see
18 node 27 in Figure 2) corresponds to node 85 in Figure 6.

19 **(Recursive) List-4:** record size of collection (1) in box 218, Figure 7. Current
20 encoding point is now start of 219, Figure 7.

21 **(Recursive) List-5: Variable-length item sub-process** chosen (arrow 77 to box 78
22 in Figure 5).

23 **Var-item-1:** one-slot offset table reserved (box 219 in Figure 7). Both **firstOffset**
24 and **nextOffset** point to start of 219. Current encoding point is now at start of box 220.

25 **Var-item-2:** iterate remaining steps for the members of the **array** node 85; this
26 comprises the single node 87.

1 **Var-item-3(0)**: record offset in slot at **nextOffset**, which is box **219**. The value is
2 current encoding point minus **firstOffset** which is 4 bytes. **NextOffset** is now at the start
3 of box **220**.

4 **Var-item-4(0)**: Residual path is [E] (see node **27** in Figure 2), so navigate from node
5 **87** to node **96** in Figure 6. The value is "cow".

6 **Var-item-5(0)**: "cow" appears in result (**220** in Figure 7).

7 **Var-item-6(0)**: current encoding point is now at **221** in Figure 7.

8 **(End variable-length item sub-process, return via arrow 77 to box 74).**

9 **(Recursive) List-6**: length of list (15 bytes) recorded in at **lengthLocation** (box **217**).

10 **(End List sub-process recursive invocation, return via arrow 79 to box 80 in Figure**
11 **5).**

12 **List-val-item-3(1)**: record offset in slot at **nextOffset**, which is box **216**. The value
13 is current encoding point minus **firstOffset** which is 27 bytes. **NextOffset** is now at the
14 start of box **217**.

15 **List-val-item-4(1)**: Recursively invoke the **list sub-process** (arrow **79** to box **74** in
16 Figure 7).

17 **(Recursive) List-1**: reserve bytes in box **221** of Figure 7 and set **lengthLocation** to
18 the start of those bytes, current encoding point and **sizeLocation** to the start of box **222**.

19 **(Recursive) List-2**: starting node is node **84** in Figure 6.

20 **(Recursive) List-3**: first homogeneous collection in the residual path [CDE] (see
21 node **27** in Figure 2) corresponds to node **86** in Figure 6.

22 **(Recursive) List-4**: record size of collection (3) in box **222**, Figure 7. Current
23 encoding point is now start of **223**, Figure 7.

24 **(Recursive) List-5: Variable-length item sub-process** chosen (arrow **77** to box **78**
25 in Figure 5).

26 **Var-item-1**: three-slot offset table reserved (boxes **223** through **225** in Figure 7).
27 Both **firstOffset** and **nextOffset** point to start of **223**. Current encoding point is now at
28 start of box **226**.

1 **Var-item-2:** iterate remaining steps for the members of the **array** node **85**; these
2 comprise of nodes **88, 89, and 90**.

3 **Var-item-3(0):** record offset in slot at **nextOffset**, which is box **223**. The value is
4 current encoding point minus **firstOffset** which is 12 bytes. **NextOffset** is now at the
5 start of box **224**.

6 **Var-item-4(0):** Residual path is **[E]** (see node **27** in Figure 2), so navigate from node
7 **88** to node **97** in Figure 6. The value is **"bird"**.

8 **Var-item-5(0):** **"bird"** appears in result (**226** in Figure 7).

9 **Var-item-6(0):** current encoding point is now at **227** in Figure 7.

10 **Var-item-3(1):** record offset in slot at **nextOffset**, which is box **224**. The value is
11 current encoding point minus **firstOffset** which is 20 bytes. **NextOffset** is now at the
12 start of box **225**.

13 **Var-item-4(1):** Residual path is **[E]** (see node **27** in Figure 2), so navigate from node
14 **89** to node **98** in Figure 6. The value is **"joe"**.

15 **Var-item-5(1):** **"joe"** appears in result (**227** in Figure 7).

16 **Var-item-6(1):** current encoding point is now at **228** in Figure 7.

17 **Var-item-3(2):** record offset in slot at **nextOffset**, which is box **225**. The value is
18 current encoding point minus **firstOffset** which is 27 bytes. **NextOffset** is now at the
19 start of box **226**.

20 **Var-item-4(2):** Residual path is **[E]** (see node **27** in Figure 2), so navigate from node
21 **90** to node **99** in Figure 6. The value is **"lime"**.

22 **Var-item-5(2):** **"lime"** appears in result (**228** in Figure 7).

23 **Var-item-6(2):** current encoding point is now at **229** in Figure 7.

24 **(End variable-length item sub-process, return via arrow 77 to box 74).**

25 **(Recursive) List-6:** length of list (39 bytes) recorded in at **lengthLocation** (box **221**).

26 **(End List sub-process recursive invocation, return via arrow 79 to box 80 in Figure**
27 **5).**

28 **(End List-valued-item sub-process, return via arrow 79 to box 74 in Figure 5).**

1 **List-6:** length of list (74 bytes) recorded in at **lengthLocation** (box 213).

2 **(End List sub-process,** return via arrow 73 to box 70 in Figure 5).

3 **Master-4(4):** field 4 is in variable length portion (box 61 in Figure 4) so serialization
4 is at current encoding point. Offset of box 61 is in box 54 in Figure 4 as shown by arrow
5 67, so the offset of the current encoding point from the start of the variable length portion
6 (124 bytes) is recorded in box 202 in Figure 7.

7 **Master-5(4): list sub-process** chosen (arrow 73 to box 74 in Figure 5).

8 **List-1:** reserve bytes in box 229 of Figure 7 and set **lengthLocation** to the start of
9 those bytes, current encoding point and **sizeLocation** to the start of box 230.

10 **List-2:** starting node is node 81 in Figure 6.

11 **List-3:** first homogeneous collection in path (see node 28 in Figure 2) corresponds to
12 node 82 in Figure 6.

13 **List-4:** record size of collection (2) in box 230, Figure 7. Current encoding point is
14 now start of 231, Figure 7.

15 **List-5: List-valued item sub-process** chosen (arrow 79 to box 80 in Figure 5).

16 **List-val-item-1:** two-slot offset table reserved (boxes 231 and 232 in Figure 7).
17 Both **firstOffset** and **nextOffset** point to start of 231. Current encoding point is now at
18 start of box 233.

19 **List-val-item-2:** iterate remaining steps for the members of the **array** node 82; these
20 are nodes 83 and 84.

21 **List-val-item-3(0):** record offset in slot at **nextOffset**, which is box 231. The value
22 is current encoding point minus **firstOffset** which is 8 bytes. **NextOffset** is now at the
23 start of box 232.

24 **List-val-item-4(0):** Recursively invoke the **list sub-process** (arrow 79 to box 74 in
25 Figure 7).

26 **(Recursive) List-1:** reserve bytes in box 233 of Figure 7 and set **lengthLocation** to
27 the start of those bytes, current encoding point and **sizeLocation** to the start of box 234.

28 **(Recursive) List-2:** starting node is node 83 in Figure 6.

1 **(Recursive) List-3:** first homogeneous collection in the residual path [CDE] (see
2 node 27 in Figure 2) corresponds to node 85 in Figure 6.

3 **(Recursive) List-4:** record size of collection (1) in box 234, Figure 7. Current
4 encoding point is now start of 235, Figure 7.

5 **(Recursive) List-5: Fixed-length item sub-process** chosen (arrow 75 to box 76 in
6 Figure 5). Steps will be iterated only once since node 85 has only one child.

7 **Fixed-item-1(0):** Residual path is [E] (see node 28 in Figure 2), so navigate from
8 node 87 to node 100 in Figure 6. The value is 4.

9 **Fixed-item-2(0):** 4 appears in result (235 in Figure 7).

10 **Fixed-item-3(0):** current encoding point is now at 236 in Figure 7.

11 **(End fixed-length item sub-process,** return via arrow 75 to box 74).

12 **(Recursive) List-6:** length of list (8 bytes) recorded in at **lengthLocation** (box 233).

13 **(End List sub-process recursive invocation,** return via arrow 79 to box 80 in Figure
14 5).

15 **List-val-item-3(1):** record offset in slot at **nextOffset**, which is box 232. The value
16 is current encoding point minus **firstOffset** which is 20 bytes. **NextOffset** is now at the
17 start of box 233.

18 **List-val-item-4(1):** Recursively invoke the **list sub-process** (arrow 79 to box 74 in
19 Figure 7).

20 **(Recursive) List-1:** reserve bytes in box 236 of Figure 7 and set **lengthLocation** to
21 the start of those bytes, current encoding point and **sizeLocation** to the start of box 237.

22 **(Recursive) List-2:** starting node is node 84 in Figure 6.

23 **(Recursive) List-3:** first homogeneous collection in the residual path [CDE] (see
24 node 28 in Figure 2) corresponds to node 86 in Figure 6.

25 **(Recursive) List-4:** record size of collection (3) in box 237, Figure 7. Current
26 encoding point is now start of 238, Figure 7.

27 **(Recursive) List-5: Fixed-length item sub-process** chosen (arrow 75 to box 76 in
28 Figure 5).

1 **Fixed-item-1(0):** Residual path is [E] (see node 28 in Figure 2), so navigate from
2 node 88 to node 101 in Figure 6. The value is -1.
3 **Fixed-item-2(0):** -1 appears in result (238 in Figure 7).
4 **Fixed-item-3(0):** current encoding point is now at 239 in Figure 7.
5 **Fixed-item-1(1):** Residual path is [E] (see node 28 in Figure 2, so navigate from
6 node 89 to node 102 in Figure 6. The value is 5.
7 **Fixed-item-2(1):** 5 appears in result (239 in Figure 7).
8 **Fixed-item-3(1):** current encoding point is now at 240 in Figure 7.
9 **Fixed-item-1(2):** Residual path is [E] (see node 28 in Figure 2), so navigate from
10 node 90 to node 103 in Figure 6. The value is 613.
11 **Fixed-item-2(2):** 613 appears in result (240 in Figure 7).
12 **Fixed-item-3(2):** current encoding point is now at 241 in Figure 7.
13 **(End fixed-length item sub-process, return via arrow 75 to box 74).**
14 **(Recursive) List-6:** length of list (16 bytes) recorded in at **lengthLocation** (box 236).
15 **(End List sub-process recursive invocation, return via arrow 79 to box 80 in Figure**
16 **5).**
17 **(End List-valued-item sub-process, return via arrow 79 to box 74 in Figure 5).**
18 **List-6:** length of list (44 bytes) recorded in at **lengthLocation** (box 229).
19 **(End List sub-process, return via arrow 73 to box 70 in Figure 5).**
20 **Master-4(5):** field 5 is in variable length portion (box 62 in Figure 4) so serialization
21 is at current encoding point. Offset of box 62 is in box 55 in Figure 4 as shown by arrow
22 68, so the offset of the current encoding point from the start of the variable length portion
23 (172 bytes) is recorded in box 203 in Figure 7.
24 **Master-5(5): list sub-process** chosen (arrow 73 to box 74 in Figure 5).
25 **List-1:** reserve bytes in box 241 of Figure 7 and set **lengthLocation** to the start of
26 those bytes, current encoding point and **sizeLocation** to the start of box 242.
27 **List-2:** starting node is node 81 in Figure 6.

1 **List-3:** first homogeneous collection in path (see node **29** in Figure 2) corresponds to
2 node **82** in Figure 6.

3 **List-4:** record size of collection (2) in box **242**, Figure 7. Current encoding point is
4 now start of **243**, Figure 7.

5 **List-5: List-valued item sub-process** chosen (arrow **79** to box **80** in Figure 5).

6 **List-val-item-1:** two-slot offset table reserved (boxes **243** and **244** in Figure 7).
7 Both **firstOffset** and **nextOffset** point to start of **243**. Current encoding point is now at
8 start of box **245**.

9 **List-val-item-2:** iterate remaining steps for the members of the **array** node **82**; these
10 are nodes **83** and **84**.

11 **List-val-item-3(0):** record offset in slot at **nextOffset**, which is box **243**. The value
12 is current encoding point minus **firstOffset** which is 8 bytes. **NextOffset** is now at the
13 start of box **244**.

14 **List-val-item-4(0):** Recursively invoke the **list sub-process** (arrow **79** to box **74** in
15 Figure 7).

16 **(Recursive) List-1:** reserve bytes in box **245** of Figure 7 and set **lengthLocation** to
17 the start of those bytes, current encoding point and **sizeLocation** to the start of box **246**.

18 **(Recursive) List-2:** starting node is node **83** in Figure 6.

19 **(Recursive) List-3:** first homogeneous collection in the residual path [**CDE**] (see
20 node **29** in Figure 2) corresponds to node **85** in Figure 6.

21 **(Recursive) List-4:** record size of collection (1) in box **246**, Figure 7. Current
22 encoding point is now start of **247**, Figure 7.

23 **(Recursive) List-5: Fixed-length item sub-process** chosen (arrow **75** to box **76** in
24 Figure 5). Steps will be iterated for nodes **88**, **89**, and **90**, the three children of node **85**.

25 **Fixed-item-1(0):** Residual path is [**E**] (see node **29** in Figure 2), so navigate from
26 node **87** to node **104** in Figure 6. The value is **true**.

27 **Fixed-item-2(0):** **true** appears in result (**247** in Figure 7).

28 **Fixed-item-3(0):** current encoding point is now at **248** in Figure 7.

1 **(End fixed-length item sub-process, return via arrow 75 to box 74).**

2 **(Recursive) List-6:** length of list (5 bytes) recorded in at **lengthLocation** (box 245).

3 **(End List sub-process recursive invocation, return via arrow 79 to box 80 in Figure**

4 **5).**

5 **List-val-item-3(1):** record offset in slot at **nextOffset**, which is box 244. The value

6 is current encoding point minus **firstOffset** which is 17 bytes. **NextOffset** is now at the

7 start of box 245.

8 **List-val-item-4(1):** Recursively invoke the **list sub-process** (arrow 79 to box 74 in

9 Figure 7).

10 **(Recursive) List-1:** reserve bytes in box 248 of Figure 7 and set **lengthLocation** to

11 the start of those bytes, current encoding point and **sizeLocation** to the start of box 249.

12 **(Recursive) List-2:** starting node is node 84 in Figure 6.

13 **(Recursive) List-3:** first homogeneous collection in the residual path [CDE] (see

14 node 29 in Figure 2) corresponds to node 86 in Figure 6.

15 **(Recursive) List-4:** record size of collection (3) in box 249, Figure 7. Current

16 encoding point is now start of 250, Figure 7.

17 **(Recursive) List-5: Fixed-length item sub-process** chosen (arrow 75 to box 76 in

18 Figure 5).

19 **Fixed-item-1(0):** Residual path is [E] (see node 29 in Figure 2), so navigate from

20 node 88 to node 105 in Figure 6. The value is **false**.

21 **Fixed-item-2(0):** **false** appears in result (250 in Figure 7).

22 **Fixed-item-3(0):** current encoding point is now at 251 in Figure 7.

23 **Fixed-item-1(1):** Residual path is [E] (see node 29 in Figure 2, so navigate from

24 node 89 to node 106 in Figure 6. The value is **false**.

25 **Fixed-item-2(1):** **false** appears in result (251 in Figure 7).

26 **Fixed-item-3(1):** current encoding point is now at 252 in Figure 7.

27 **Fixed-item-1(2):** Residual path is [E] (see node 29 in Figure 2), so navigate from

28 node 90 to node 107 in Figure 6. The value is **true**.

1 **Fixed-item-2(2): true** appears in result (**252** in Figure 7).

2 **Fixed-item-3(2):** current encoding point is now at **253** in Figure 7.

3 **(End fixed-length item sub-process,** return via arrow **75** to box **74**).

4 **(Recursive) List-6:** length of list (7 bytes) recorded in at **lengthLocation** (box **248**).

5 **(End List sub-process recursive invocation,** return via arrow **79** to box **80** in Figure
6 **5).**

7 **(End List-valued-item sub-process,** return via arrow **79** to box **74** in Figure 5).

8 **List-6:** length of list (32 bytes) recorded in at **lengthLocation** (box **241**).

9 **(End List sub-process,** return via arrow **73** to box **70** in Figure 5).

10 **Master-4(6):** field 6 is in variable length portion (box **63** in Figure 4) so serialization
11 is at current encoding point. Offset of box **63** is in box **56** in Figure 4 as shown by arrow
12 **69**, so the offset of the current encoding point from the start of the variable length portion
13 (208 bytes) is recorded in box **204** in Figure 7.

14 **Master-5(6): Non-list sub-process** chosen (arrow **71** to box **74** in Figure 5).

15 **Non-list-1:** navigate from node **81** to **108** in Figure 6, based on the path recorded in
16 node **30** in Figure 2. The value is **-12**.

17 **Non-list-2: -12** appears in result (**253** in Figure 7).

18 **Non-list-3:** current encoding point now at start of **254** in Figure 7.

19 **(End Non-list sub-process,** return via arrow **71** to box **70** in Figure 5).

20 **Master-4(7):** field 7 is in variable length portion (box **64** in Figure 4), so serialization
21 is at the current encoding point. No offset to record.

22 **Master-5(7): non-list sub-process** chosen (arrow **71** to box **72** in Figure 5).

23 **Non-list-1:** navigate from node **81** to **109** in Figure 6, based on the path recorded in
24 node **31** in Figure 2. The value is **"pear"**.

25 **Non-list-2: "pear"** appears in result (**254** in Figure 7).

26 **Non-list-3:** current encoding point now at end of **254** in Figure 7.

27 **(End Non-list sub-process,** return via arrow **71** to box **70** in Figure 5).

28 **Master-6:** current encoding point minus start of byte stream is length of byte stream.

1 **Master process ends. Serialization process ends.**

2 **THE RANDOM ACCESS PROCESS**

3 To fulfill the promise of the invention, the random access process supports two operations.
4 Both are accomplished without deserializing the byte stream as a whole.

5 1. Retrieve a single scalar value from the byte stream, given only (1) the field number (from
6 the layout) to which the value corresponds and (2) the index positions in any homogeneous
7 collections within which the value is enclosed. This is accomplished in near-constant time.

8 2. Given a "table" (represented in the schema as a **list of tuples** and represented in the
9 in-memory representation as a homogeneous collection of heterogeneous collections), scan a
10 column of that table within the byte stream to determine the index matched by a particular key
11 value. The table row is designated by (1) the field number (from the layout) to which the values
12 making up the column correspond and (2) the index positions in any homogeneous collections
13 within which all of the values comprising the column are enclosed. This is accomplished in
14 time proportional to the number of rows in the table but nearly insensitive to the number of
15 columns or other aspects of information structure complexity.

16 The description of the random access process is a description of how these two operations are
17 accomplished. This is followed by an example that illustrates both operations.

18 At the start of either operation, the schema tree representation of a particular schema is
19 available, along with the layout computed from that schema tree representation, and a byte
20 stream resulting from serialization (at some earlier time) of an in-memory representation
21 conforming to that schema tree representation. The in-memory representation itself is not
22 available.

23 **Retrieving Scalar Values**

1 Available at the start of this operation are a byte stream, a layout, a schema tree
2 representation, a field number whose value is of interest, and zero or more non-negative integer
3 position numbers. If the field number has more than one scalar value in the byte stream, that will
4 be because the schema leaf node to which it corresponds has **list** elements in its path. For each
5 such list element, one position number corresponds and indicates a position in the homogeneous
6 collection in the in-memory representation from which the byte stream was serialized. One way
7 that these position numbers will have been obtained is by employing the operation of scanning
8 table columns described below as part of this invention. The position numbers together uniquely
9 determine a scalar value. If the number of supplied position numbers does not match the number
10 of **list** elements in the path, the request does not properly designate a scalar value and the
11 operation fails. Otherwise, the operation comprises 7 steps.

12 **Step 1.** The field number is used to consult the layout. This says whether the value is in the
13 fixed-length or variable-length part of the byte stream and, if it is in the variable-length portion,
14 how to compute its offset. If the value is in the fixed length portion, its offset is already known.
15 The value is retrieved and the operation ends. Otherwise, remaining steps are executed..

16 **Step 2.** The offset of the field within the variable-length portion is determined. Details
17 depend on the layout style.

18 **Layout style 1.** The offset of the field is read from a slot in the fixed-length portion of the
19 byte stream that corresponds to the field number.

20 **Layout style 2.** There are two substeps.

21 **Sub-step a.** Determine the field whose field number is equal to or less than that of the field
22 of interest and whose offset is recorded in the fixed-length byte stream portion. Read that offset.
23 If there is no such field, use as the offset the start of the variable-length byte stream portion.

24 **Sub-step b.** If sub-step (a) provided the offset of the field, step 2 is accomplished.
25 Otherwise, add a precomputed increment to the offset of sub-step (a) to get the offset of the field
26 of interest. This precomputed increment will be the combined lengths of fixed-length fields
27 preceding the field in the byte stream starting with the one whose offset is recorded.

1 While processing for layout style 2 sounds more complex, only one offset is read from the
2 byte stream and all other information necessary to accomplish the action efficiently is
3 precomputed and part of the layout. So, this, too, is a constant-time operation.

4 **Step 3.** If there are no position numbers, the value is read from the byte stream at the offset
5 computed in step 2 and the operation completes. Otherwise, remaining steps are executed.

6 **Step 4.** Iterate through the supplied position numbers, performing steps 5 and 6 on each
7 position number.

8 **Step 5 (repeated by step 4).** This step adds 8 to the "previous offset" which is either the
9 offset computed by step 2 or the offset computed by the previous iteration of step 6. Adding 8
10 bytes skips over the length and size fields that are present at the start of every list.

11 **Step 6 (repeated by step 4).** This step computes a new offset from the offset computed by
12 step 5. There are two cases, determined as follows. If the iteration has reached the last position
13 number and the field has a fixed length encoding, we perform the **fixed** case. Otherwise, we
14 perform the **varying** case.

15 **Fixed case.** The position number is multiplied by the size in bytes of the fixed encoding for
16 the field's data type. This result is added to the offset computed in step 5.

17 **Varying case.** The position number is multiplied by four. This result is added to the offset
18 computed in step 5, yielding the offset of a slot in the list's offset table. An offset is read from
19 that slot in the offset table and added to the offset computed in step 5.

20 **Step 7.** A value is read from the byte stream at the offset computed by the last iteration of
21 **step 6.** That is the desired value and the operation completes.

22 **Scanning Table Columns**

1 Available at the start of this operation are a byte stream, a layout, a schema tree
2 representation, a key item to be matched, a field number whose values are to be scanned for a
3 match, and zero or more position numbers. The field number must be one that designates more
4 than one scalar value in the byte stream because the schema leaf node to which it corresponds has
5 **list** elements in its path (otherwise, the operation fails). For each list element in the field's path,
6 except for the last, one of the supplied position numbers corresponds and indicates a position in
7 the homogeneous collection in the in-memory representation from which the byte stream was
8 serialized (there will be zero position numbers if the path has only one **list** element). If the
9 number of supplied position numbers does not match the number of **list** elements in the field's
10 path minus one, the request does not properly designate a table column with scalar values and the
11 operation fails. Otherwise, it comprises 9 steps.

12 **Step 1.** The field number is used to consult the layout. This says how to find the field in the
13 variable-length portion of the byte stream (it must be in that portion, since it is a list).

14 **Step 2.** The offset of the field within the variable-length portion is determined. Details
15 depend on the layout style.

16 **Layout style 1.** The offset of the field is read from a slot in the fixed-length portion of the
17 byte stream corresponding to the field number.

18 **Layout style 2.** There are two substeps.

19 **Sub-step a.** Determine the leaf node whose field number is equal to or less than that of the
20 field of interest and whose offset is recorded in the fixed-length byte stream portion. Read that
21 offset. If there is no such field, use as the offset the start of the variable-length byte stream
22 portion.

23 **Sub-step b.** If sub-step (a) provided the offset of the field, step 2 is accomplished.
24 Otherwise, add a precomputed increment to the offset of sub-step (a) to get the offset of the field
25 of interest. This precomputed increment will be the combined lengths of fixed-length fields
26 preceding the field in the byte stream.

27 **Step 3.** Iterate through the position numbers, if any, performing steps 4 and 5 on each
28 position number. If there are no position numbers, skip steps 4 and 5.

1 **Step 4 (repeated by step 3).** This step adds 8 to the "previous offset" which is either the
2 offset computed by step 2 or the offset computed by the previous iteration of step 5. Adding 8
3 bytes skips over the length and size fields that are present at the start of every list.

4 **Step 5 (repeated by step 3).** This step computes a new offset from the offset computed by
5 step 4. The position number is multiplied by four. This result is added to the offset computed in
6 step 4, yielding the offset of a slot in the list's offset table. An offset is read from that slot in the
7 offset table and added to the offset computed in step 4.

8 **Step 6.** The offset now points to the list that is to be scanned. The size of the list (number
9 of elements) is read from a point four bytes after the start of the list (skipping over the length
10 field). That gives the number of rows in the table. The offset is incremented by 8 to skip both
11 the length field and the size field.

12 **Step 7.** If the field has a scalar data type that has a variable length encoding, multiply the
13 size by 4 and add this to the offset. That skips over the offset table and gives the offset of the
14 actual data in the list. Otherwise, there is no offset table and the offset computed in **step 6** is
15 used unchanged.

16 **Step 8.** Iterate through the items of data in the list, comparing each item to the key item, and
17 stopping on a match or after visiting the entire list as given by the size which was read in **step 6**.
18 Recall that all scalar value encodings must provide a way of determining where they start and
19 end so that such sequential scanning is possible.

20 **Step 9.** If step 8 terminated with a match, return the index position of the matched item.
21 Otherwise, indicate a "not matched" exception.

22 **Example.**

23 Suppose the application, having the byte stream depicted in Figure 7, wants to retrieve one
24 of the boolean values of field number is 5, corresponding to the key value "dog" for field 2
25 (string) and the key value "joe" for field 3 (string).

1 The way the application designer would think of this operation is best understood with
2 reference to Figure 2, which shows the schema tree representation with field numbers added.
3 The application wants to find the member of the **list** shown as node **19** that has "**dog**" as its
4 value for node **26** (field 2), then, in that same list member, find the member of the list shown as
5 node **21** that has "**joe**" as its value for node **27** (field 3), then, in that same list member, retrieve
6 the value of node **29** (field 5).

7 To scan the values of field 2 for the key value "**dog**" the application supplies both the field
8 number 2 and the key value "**dog**" as inputs to the operation of scanning table columns.

9 **Step 1.** The layout (see Figure 4) says that field 2 is in the variable-length message portion
10 (byte range **59**), using the offset to field 1 recorded in slot 0 of the fixed portion (slot **52** and
11 arrow **65**).

12 **Step 2.** The offset in slot 0 of the fixed portion (8 bytes) is read from the byte stream (**200** in
13 Figure 7) and the length of field 1 (4 bytes) is added to it to provide the offset to field 2. This is
14 12 bytes, which when added to the start of the variable length portion at **205** in Figure 7,
15 computes the start of byte range **207** in Figure 7.

16 **Step 3.** As there are no position numbers, steps 4 and 5 will be skipped.

17 **Step 4 / step 5:** Skipped (iterated zero times by step 3).

18 **Step 6:** Adding 4 bytes to the offset of byte range **207** yields byte range **208** and the size of
19 the list (2) is read from there. We skip to the beginning of the offset table (box **209** in Figure
20 7).

21 **Step 7:** The offset table (8 bytes as computed by this step) is skipped over and we are now
22 pointing to the first element in the list ("**charles**", which is **211** in Figure 7).

23 **Step 8:** Iterate through the two items looking for a match on "**dog**" which occurs at index 1
24 (it is matched as the second of two items in the list).

25 **Step 9:** Return index 1.

26 The application now knows that the nested table corresponding to key value "**dog**" is at
27 position 1 in the homogeneous collection represented by node **19** in Figure 2.

1 To scan the values of field number 3 for the key value "joe", the operation for scanning
2 tables is invoked again, this time with field number 3, key item "joe", and a single position
3 number 1 determined in the previous scanning operation.

4 **Step 1.** The layout (see Figure 4) says that field number 3 is in the variable-length message
5 portion (area 60), with its offset recorded in slot 1 of the fixed portion (slot labeled 53, and arrow
6 66).

7 **Step 2.** The offset in slot 1 of the fixed portion (46 bytes) is read (201 in Figure 7) to give
8 the offset to field 3 (213 in Figure 7).

9 **Step 3.** There will be one iteration of steps 4 and 5 with the position number 1.

10 **Step 4.** The length and size fields (213 and 214) are skipped over, yielding the offset of the
11 start of 215.

12 **Step 5.** The second offset table entry (position number 1) slot is read (27 bytes, read from
13 box 216 in Figure 7) and the value added to the offset of step 4, yielding the offset of the second
14 list (221 in Figure 7).

15 **Step 6.** The size field (3) is read (222 in Figure 7) and it and the length field are skipped over
16 yielding the offset of 223 in Figure 7.

17 **Step 7.** The offset table is skipped over yielding the offset of 226 in Figure 7.

18 **Step 8.** Iterate through the items looking for a match on "joe" which occurs at index 1.

19 **Step 9.** Index 1 is returned.

20 The application now knows that the desired value in field 5 is at positions 1 and 1,
21 respectively, in the nested homogeneous collections represented by nodes 19 and 21 in Figure 2.
22 The scalar value random access operation is invoked with that information.

23 **Step 1.** The layout (see Figure 4) says that field 5 is in the variable-length message portion
24 (62 in Figure 4) with its offset stored in the fourth slot of the fixed portion (slot 55 and arrow 68
25 in Figure 4).

26 **Step 2.** The offset in the fourth slot of the fixed portion (172 bytes) is read (203 in Figure 7)
27 to give the offset to field 5 (241 in Figure 7).

1 **Step 3.** As there are position numbers remaining to be processed, a value cannot be returned
2 at this point so the operation continues.

3 **Step 4.** The position numbers 1 and 1 will each cause an iteration of steps 5 and 6.

4 **Step 5(0).** 8 is added to the offset of **241**, yielding the offset of **243**.

5 **Step 6(0).** The offset is adjusted by 17 bytes, read from the second offset table slot (**244** in
6 Figure 7), yielding the offset of **248**.

7 **Step 5(1).** 8 is added to the offset of **248**, yielding the offset of **250**.

8 **Step 6(1).** The offset is increased by the length of a boolean, times 1 and now addresses the
9 value (false) in slot **251**, which is the desired value.

10 **Step 7.** The desired value is read from the message and returned.

11 **THE SCHEMA REORGANIZATION PROCESS (OPTIONAL)**

12 When a schema contains variants, there is an alternative to truncating the schema at the
13 variant nodes and changing the variants to dynamic type. The alternative, which is based on
14 known results in type isomorphism, turns a single schema into several schemas, each describing
15 one case of a top-level variant, where that variant is the result of distributing tuples over variants
16 to the greatest extent possible.

17 Figure 8 shows a schema that contains two variants (nodes **301** and **307**) as well as tuples
18 (**300** and **306**), a list (**305**) and scalar types (**302** through **304**, and **308** through **310**). If this
19 schema were to be truncated in the usual way, both variants would be replaced by dynamic type
20 leaf nodes. In a schema as simple as this, such a truncation will probably not harm efficiency to
21 a serious extent. But, each dynamic type node requires a recursive use of the invention and each
22 recursive use of the invention adds overhead to the ultimate goal of accessing individual scalar
23 values from the byte stream.

24 As an alternative, the tuple **300** can be distributed over the variant **301** and the tuple **306** can
25 be distributed over the variant **307**, resulting in the four schemas shown in Figure 9. The
26 contents of Figure 9 will be explained after describing the general algorithm.

27 The algorithm for distributing a tuple over a variant comprises the following nine steps.

1 **Step 1.** Find an occurrence in the schema where a **tuple** is a child of another **tuple** or a
2 **variant** is a child of another **variant**. If any such case is found, remove the child **tuple** and
3 make its children into direct children of the parent **tuple** or remove the child **variant** and make
4 its children into direct children of the parent **variant**.

5 **Step 2.** Repeat step 1 until it can no longer be applied.

6 **Step 3.** Find an occurrence in the schema where a **variant** is a child of a **tuple**. If any such
7 case is found, perform steps 4 through 6 using that **variant** and **tuple**.

8 **Step 4.** Make a new tuple is comprised of the **variant's** first child and all children of the
9 **tuple** other than the **variant**.

10 **Step 5.** Repeat **step 4** for all of the remaining children of the **variant**, resulting in as many
11 new **tuples** as there were children of the **variant**.

12 **Step 6.** Form a new **variant** whose children are the new **tuples** created in steps 4 and 5.
13 Replace the original **tuple** and all of its descendants in the schema tree with the new **variant** and
14 all of its descendants.

15 **Step 7.** Repeat steps 1 through 6 until none of them are applicable.

16 The result of applying this algorithm to a schema tree representation whose interior nodes
17 comprise only of tuples and variants is a schema tree representation whose sole variant (if any) is
18 the root node (there will be no variants at all if there were none to begin with). The invention
19 can then be applied by discarding the variant root node and treating each case of the variant as a
20 different schema for the purpose of using the invention.

21 The result of applying this algorithm to a schema tree representation with list nodes as well as
22 tuples and variants may retain variants in the result that are not at the root. These will always be
23 the direct children of lists, and they arise because the algorithm's steps will never bring a variant
24 and tuple into direct parent-child relationship when a list intervenes. These variants under lists
25 must still be replaced by dynamic type nodes as described earlier.

26 Figure 9 shows the result of applying the schema reorganization process to the example
27 schema of Figure 8. The schema rooted at node **350** represents the original schema when the
28 variant **301** in Figure 8 takes on an integer value. It is a tuple with children **351**, **352**, and **353**,

1 corresponding to the original nodes 302, 303, and 304 (respectively) in Figure 8. Note that the
2 child of the list node 353 is now a dynamic node 354, which represents a truncation because the
3 algorithm for distributing tuple 306 over variant 307 created a new variant node at this point.

4 Similarly, the schema is comprised of nodes 355 through 359 represents the original schema
5 when the variant 301 takes on a boolean value.

6 The schema is comprised of nodes 360 through 362 represents the fragment of the original
7 schema whose local root is 306 when variant 307 takes on an integer value. It is used to encode
8 each dynamic type value of nodes 354 or 359 that contains an integer. Similarly, the schema is
9 comprised of nodes 363 through 365 represents the same fragment of the original schema whose
10 local root is 306 when variant 307 takes on a boolean value. It is used to encode each dynamic
11 type value of nodes 354 or 359 that contains a boolean.

12 APPARATUS IMPLEMENTATION

13 The present invention includes an apparatus performing methods of this invention. In an
14 example embodiment, the apparatus comprises a serializer/deserializer for a byte stream form of
15 an information structure, said information structure having a schema and an in-memory
16 representation, said schema having a schema tree representation with a plurality of schema
17 nodes, said schema nodes including at least one leaf and at least one interior node. The
18 serializer/deserializer comprising: a processor for computing a layout from the schema tree
19 representation by depth-first enumeration of leaf nodes of the schema; a serializer for serializing
20 the byte stream from the in-memory representation while grouping together all scalar items from
21 the in-memory representation corresponding to each schema node; and a selective de-serializer
22 for accessing information from the byte stream by using the layout and offset calculations.

23 In some embodiments of the apparatus, the processor comprises a module for establishing a
24 fixed length portion of the byte stream, the fixed length portion having a slot for each enumerated
25 schema leaf node; and for establishing a varying length portion of the byte stream following the
26 fixed length portion, the varying length portion having successive areas for any information items
27 requiring varying length encoding.

1 In other embodiments of the apparatus, the processor comprises a module for establishing a
2 fixed length portion of the byte stream, the fixed length portion having a slot for each enumerated
3 schema leaf node having a predecessor in the depth-first numbering requiring varying length
4 encoding; and for establishing a varying length portion of the byte stream following the fixed
5 length portion, the varying length portion having successive areas for each enumerated schema
6 node.

7 In some cases, the serializer comprises: a reconciling module to determine a correspondence
8 between the in-memory representation and the schema tree representation; an initialization
9 module to initialize the byte stream by reserving a fixed length portion and pointing to a
10 beginning of a variable length portion; a lookup module to retrieve a location in the byte stream
11 for an element of the in-memory representation information corresponding to a first schema leaf
12 node in depth first order from the layout; and a converter to convert the element to bytes in the
13 byte stream according to a number of elements corresponding to the schema leaf node, wherein
14 all schema leaf nodes are retrieved and converted in depth-first order.

15 In some embodiments the converter comprises a recorder to record a nested list of tuples in
16 column order rather than row order, resulting in a set of nested lists, and/or the converter
17 precedes each list of varying length items with an offset table allowing any element of said each
18 list to be reached in constant time from a head of said each list.

19 In some embodiments, the selective de-serializer scans a list of key values representing a
20 table column serialized within the byte stream to determine an index position, and uses the index
21 position in conjunction with offset calculations and offset tables serialized at the starts of lists
22 within the byte stream to find information in lists representing non-key table columns.

23 In some embodiments, the schema tree representation is derived from a schema graph
24 representation by truncating recursive definitions and variants and replacing them with leaf nodes
25 of dynamic type, and/or a preliminary reorganization of the schema is performed to distribute
26 tuples over variants prior to carrying out the remaining steps.

27 Variations described for the present invention can be realized in any combination desirable
28 for each particular application. Thus particular limitations, and/or embodiment enhancements

1 described herein, which may have particular advantages to a particular application need not be
2 used for all applications. Also, not all limitations need be implemented in methods, systems
3 and/or apparatus including one or more concepts of the present invention.

4 The present invention can be realized in hardware, software, or a combination of hardware
5 and software. A visualization tool according to the present invention can be realized in a
6 centralized fashion in one computer system, or in a distributed fashion where different elements
7 are spread across several interconnected computer systems. Any kind of computer system - or
8 other apparatus adapted for carrying out the methods and/or functions described herein - is
9 suitable. A typical combination of hardware and software could be a general purpose computer
10 system with a computer program that, when being loaded and executed, controls the computer
11 system such that it carries out the methods described herein. The present invention can also be
12 embedded in a computer program product, which comprises all the features enabling the
13 implementation of the methods described herein, and which - when loaded in a computer system
14 - is able to carry out these methods.

15 Computer program means or computer program in the present context include any
16 expression, in any language, code or notation, of a set of instructions intended to cause a system
17 having an information processing capability to perform a particular function either directly or
18 after conversion to another language, code or notation, and/or reproduction in a different material
19 form.

20 Thus the invention includes an article of manufacture which comprises a computer usable
21 medium having computer readable program code means embodied therein for causing a function
22 described above. The computer readable program code means in the article of manufacture
23 comprises computer readable program code means for causing a computer to effect the steps of a
24 method of this invention. Similarly, the present invention may be implemented as a computer
25 program product comprising a computer usable medium having computer readable program code
26 means embodied therein for causing a a function described above. The computer readable
27 program code means in the computer program product comprising computer readable program
28 code means for causing a computer to effect one or more functions of this invention.

1 Furthermore, the present invention may be implemented as a program storage device readable by
2 machine, tangibly embodying a program of instructions executable by the machine to perform
3 method steps for causing one or more functions of this invention.

4 It is noted that the foregoing has outlined some of the more pertinent objects and
5 embodiments of the present invention. This invention may be used for many applications. Thus,
6 although the description is made for particular arrangements and methods, the intent and concept
7 of the invention is suitable and applicable to other arrangements and applications. It will be clear
8 to those skilled in the art that modifications to the disclosed embodiments can be effected
9 without departing from the spirit and scope of the invention. The described embodiments ought
10 to be construed to be merely illustrative of some of the more prominent features and applications
11 of the invention. Other beneficial results can be realized by applying the disclosed invention in a
12 different manner or modifying the invention in ways known to those familiar with the art.